

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação



Trabalho Prático 1 - 1º Sem 2019

SCC0217 - Linguagens de Programação e Compiladores
Prof. Diego Raphael Amancio

Integrantes do grupo

Nº USP

Alyson Matheus Maruyama Nascimento	8532269
Wallace Cruz de Souza	9779392
Gabriel Matheus Bezerra Alves de Carvalho	9779429
Lucas Alberico Macedo	9293585

Introdução

O trabalho consiste no desenvolvimento de um analisador léxico para a linguagem LALG (pascal simplificado) incluindo o tratamento de erros. Com isso, tendo um programa em LALG como entrada, o analisador léxico deve apresentar os conjuntos <cadeia, token>, assim como os erros respectivos de cada cadeia.

Decisões de Projeto

Ferramentas: o analisador léxico foi implementado utilizando o flex para a declaração das expressões regulares e tokens. Uma função principal em linguagem C analisa todo o arquivo de entrada chamando o analisador léxico diversas vezes.

Par cadeia-token: o par cadeia-token é representado pela palavra reservada da linguagem LALG seguida de seu correspondente símbolo. Por exemplo, para a palavra reservada “:”, temos o conjunto <: , DOIS_PONTOS>. Essa escolha foi feita para facilitar a legibilidade e entendimento do par.

Implementação da tabela de palavras reservadas: foi utilizado uma tabela hash para armazenar as palavras reservadas pela linguagem. A escolha dessa estrutura deve-se ao fato de sua leitura possuir ordem de grandeza constante ($O(1)$), mantendo o alto desempenho independentemente do tamanho da linguagem. Para esse problema foi utilizada uma tabela de 25 posições, permitindo a extensão da linguagem (que já foi estendida com o comando: for x to y). Essa hash, definida no arquivo `regex.l`, é implementada como um vetor estático, isso para que seu desempenho seja melhor, e armazena a palavra reservada da linguagem, a mensagem de retorno do analisador e um valor de retorno. Esse último é posteriormente utilizado no arquivo `main.c` que simula o analisador sintático.

Erros: os erros (número mal formado, fim de arquivo inesperado, tamanho de identificador muito extenso, identificador mal formado, etc.) foram devidamente identificados, porém são tratados de forma genérica, deixando para a análise sintática o tratamento específico dos erros. Portanto o tratamento de erro tem o seguinte comportamento:

- Identificador: verificado se começa com um dos seguintes caracteres `[_at#$$%&]`
 - Se começar com número, tratado como número mal formatado;
 - Caso o erro encontra-se no meio do identificador: `test@ando`. Retorna-se um identificador `test` e um identificador mal formado `@ando`.
- Número: verifica se existe um dos seguintes caracteres `[_@#$$%&a-zA-Z]` em alguma parte depois de um número:
 - Caso seja um número no formato `12a` ou `1a2`:
 - É retornado um número mal formado (`12a` ou `1a2`).
 - Caso seja um número no formato `12a.344`:

- São retornados um número mal formatado (12a), um ponto (.) e um número inteiro (344).
- Caso seja um número no formato 12a.a344, retorna-se:
 - Um número mal formatado (12a), um ponto (.) e um identificador (a344).
- Caso seja um número no formato 12.a34, ou 12.3a4, ou 12.34a:
 - É retornado um número mal formatado (12.a34 ou 12.3a4 ou 12.34a).

Por fim, esses tokens serão analisados e seus erros melhor identificados na fase de análise sintática.

Arquivo: todos os pares <cadeia, token> são armazenados em arquivo (token.txt), o qual também contém os erros para as entradas inválidas e/ou não reconhecidas.

Compilação

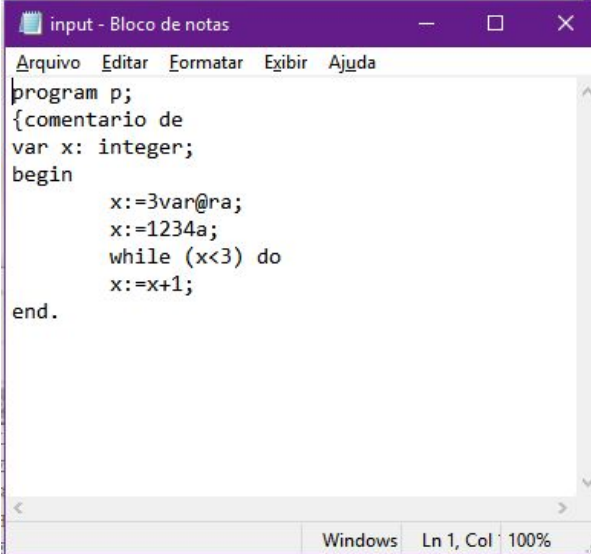
Considerando um arquivo `input.txt` contendo o programa-fonte em LALG que servirá de entrada para o analisador léxico, o arquivo flex `regex.l` e o arquivo em C `main.c` contendo a função principal, a compilação deve ser realizada da seguinte maneira nos sistemas operacionais:

Ubuntu: utilizar o comando `make` para a compilação e `make run` para a execução com o arquivo `input.txt` como entrada padrão. O Makefile está incluso no arquivo zip/rar entregue. Alternativamente, pode-se compilar com os comandos `flex regex.l` e `gcc -g main.c lex.yy.c -lfl -o lexical` e executar com `./lexical <input.txt`

Windows: utilizar o comando `flex regex.l` e `gcc main.c lex.yy.c -o lexical`. Rodar o programa com `lexical<input.txt`

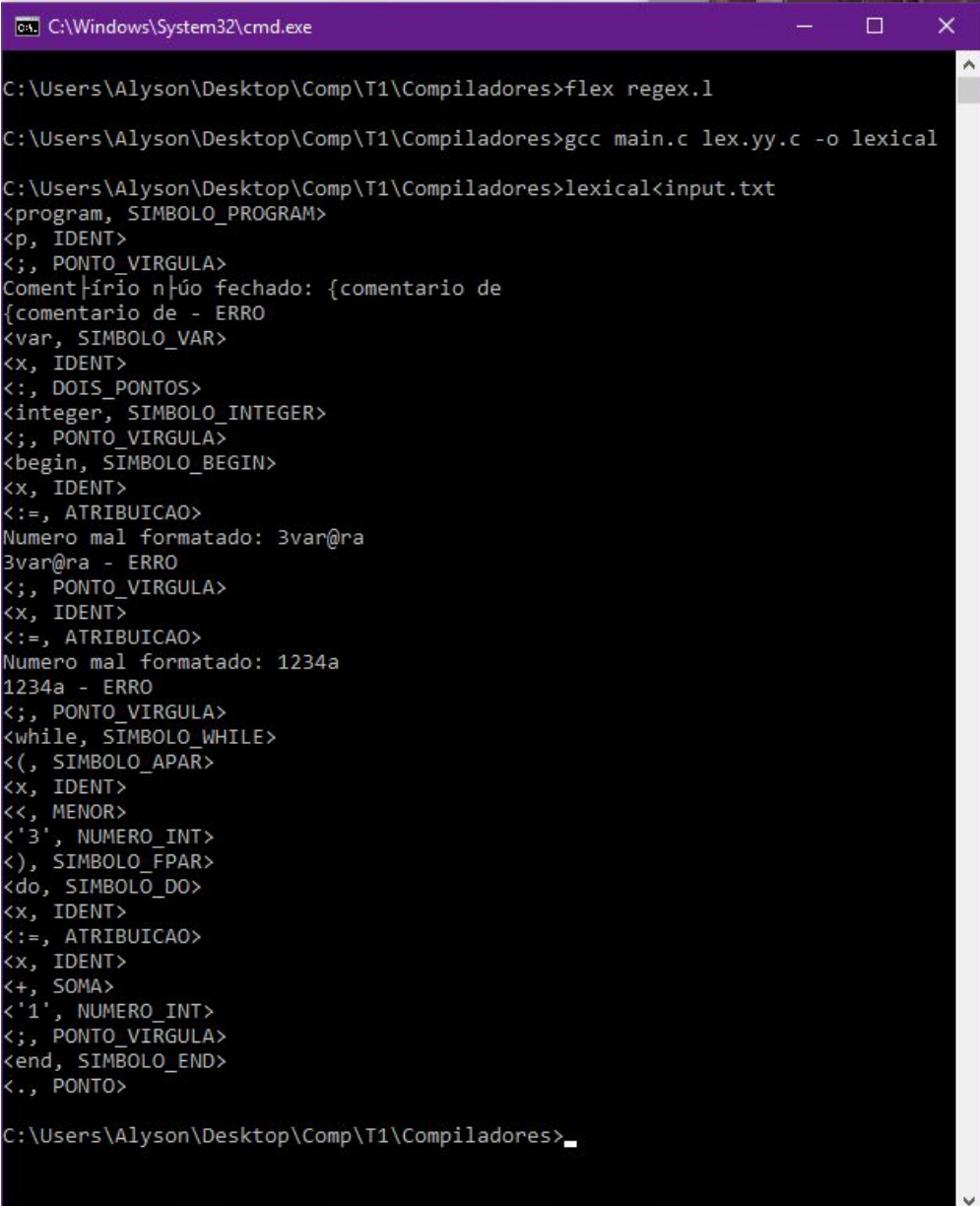
Exemplo de execução

input.txt:



```
Arquivo  Editar  Formatar  Exibir  Ajuda
program p;
{comentario de
var x: integer;
begin
    x:=3var@ra;
    x:=1234a;
    while (x<3) do
    x:=x+1;
end.
```

compilação
e
execução:



```
C:\Windows\System32\cmd.exe


C:\Users\Alyson\Desktop\Comp\T1\Compiladores>flex regex.l

C:\Users\Alyson\Desktop\Comp\T1\Compiladores>gcc main.c lex.yy.c -o lexical

C:\Users\Alyson\Desktop\Comp\T1\Compiladores>lexical<input.txt
<program, SIMBOLO_PROGRAM>
<p, IDENT>
<;, PONTO_VIRGULA>
Comentário não fechado: {comentario de
{comentario de - ERRO
<var, SIMBOLO_VAR>
<x, IDENT>
<:, DOIS_PONTOS>
<integer, SIMBOLO_INTEGER>
<;, PONTO_VIRGULA>
<begin, SIMBOLO_BEGIN>
<x, IDENT>
<:=, ATRIBUICAO>
Numero mal formatado: 3var@ra
3var@ra - ERRO
<;, PONTO_VIRGULA>
<x, IDENT>
<:=, ATRIBUICAO>
Numero mal formatado: 1234a
1234a - ERRO
<;, PONTO_VIRGULA>
<while, SIMBOLO_WHILE>
<(, SIMBOLO_APAR>
<x, IDENT>
<<, MENOR>
<'3', NUMERO_INT>
<), SIMBOLO_FPAR>
<do, SIMBOLO_DO>
<x, IDENT>
<:=, ATRIBUICAO>
<x, IDENT>
<+, SOMA>
<'1', NUMERO_INT>
<;, PONTO_VIRGULA>
<end, SIMBOLO_END>
<., PONTO>

C:\Users\Alyson\Desktop\Comp\T1\Compiladores>_
```

tokens.txt (gerado):



```
tokens - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda
program - SIMBOLO_PROGRAM
p - IDENT
; - PONTO_VIRGULA
{comentario de - ERRO
var - SIMBOLO_VAR
x - IDENT
: - DOIS_PONTOS
integer - SIMBOLO_INTEGER
; - PONTO_VIRGULA
begin - SIMBOLO_BEGIN
x - IDENT
:= - ATRIBUICAO
3var@ra - ERRO
; - PONTO_VIRGULA
x - IDENT
:= - ATRIBUICAO
1234a - ERRO
; - PONTO_VIRGULA
while - SIMBOLO_WHILE
( - SIMBOLO_APAR
x - IDENT
< - MENOR
3 - NUMERO_INT
) - SIMBOLO_FPAR
do - SIMBOLO_DO
x - IDENT
:= - ATRIBUICAO
x - IDENT
+ - SOMA
1 - NUMERO_INT
; - PONTO_VIRGULA
end - SIMBOLO_END
. - PONTO
```

Windows Ln 1, Col 100%